Sébastien Boson (2ème IL)
https://github.com/sebastie-boson/Teaching-HEIGVD-RES-2016-Labo-Semaine-02

Laboratoire de la semaine 2 (29 février)

Conditions de l'expérience

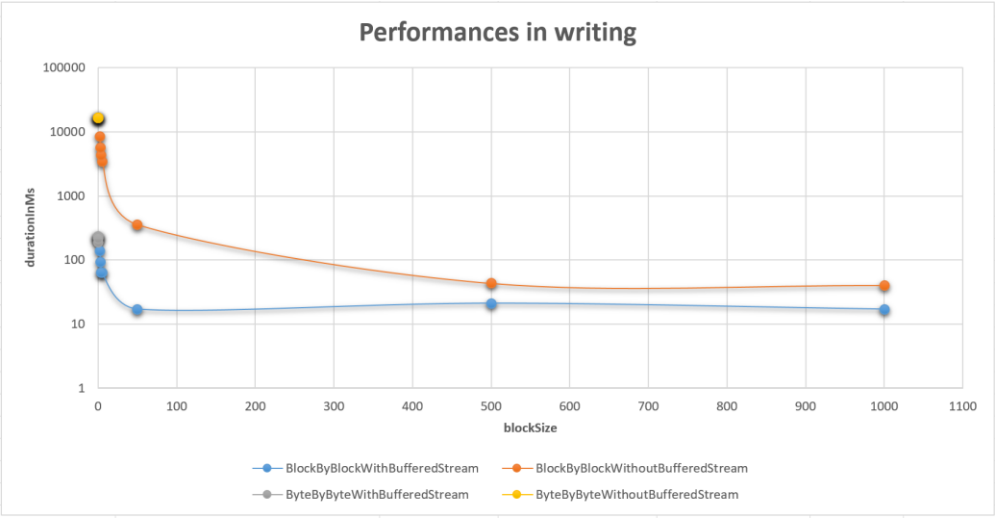
Cette expérience cherche à montrer que les performances peuvent être grandement améliorées lors de l'écriture et de la lecture de données en Java en utilisant simplement des tampons (« buffers » en anglais). Pour le déroulement de l'expérience, nous utilisons un programme écrit en Java (fichier « BufferedIOBenchmark.java ») qui va nous permettre d'écrire des données de tests dans différents fichiers et ensuite de relire les données de ces mêmes fichiers. Nous mettons ainsi en évidence le temps nécessaire pour l'écriture et la lecture de ces données suivant différentes stratégies. Une première stratégie consiste à utiliser des tampons avec une taille de bloc définie (nombre d'octets par bloc), une deuxième consiste à utiliser des tampons mais avec une taille de bloc égale à 0 (les données sont écrites et lues octet après octet), la troisième et quatrième stratégies consistent en la même chose que les deux premières mais en n'utilisant aucun tampon. L'expérience permet de mettre ainsi en évidence que l'écriture et la lecture des données est bien plus rapide en utilisant des tampons.

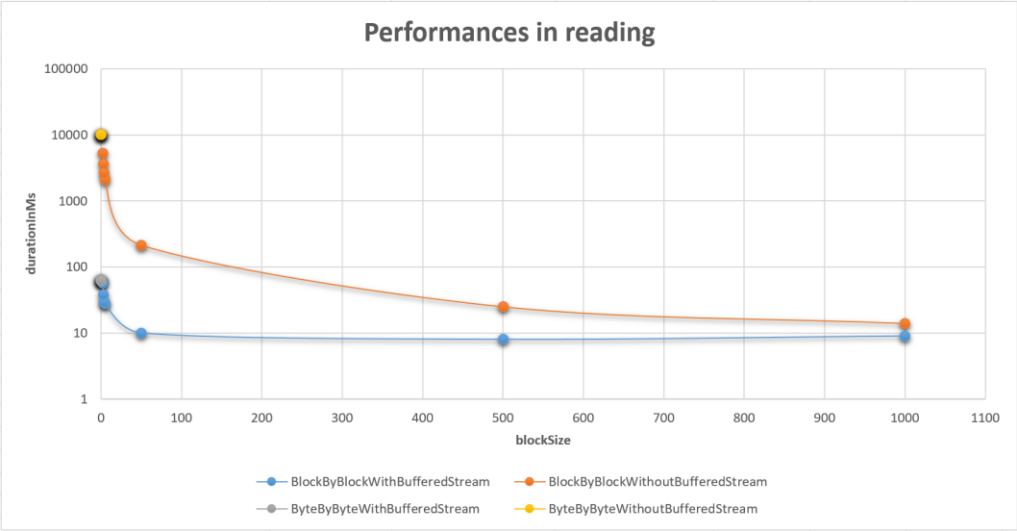
Présentation des mesures

Notre programme permet également d'enregistrer dans un fichier au format « csv » les différents temps obtenus pour toutes les stratégies selon l'écriture et la lecture des données. Ci-dessous, nous avons les deux tableaux contenus dans le fichier « csv » généré ainsi que les deux graphiques, se rapportant aux deux tableaux, que nous avons dessinés en utilisant un tableur (Excel). Il est important de préciser que l'échelle utilisée pour représenter la durée en ms est logarithmique (en base 10) pour avoir de meilleurs graphiques. De plus, la taille du fichier dans lequel les données sont écrites est fixée et ne change pas (pour pouvoir effectuer les comparaisons).

operation	strategy	blockSize	fileSizeInBytes	durationInMs
WRITE	BlockByBlockWithBufferedStream	1000	10485760	17
WRITE	BlockByBlockWithBufferedStream	500	10485760	21
WRITE	BlockByBlockWithBufferedStream	50	10485760	17
WRITE	BlockByBlockWithBufferedStream	5	10485760	63
WRITE	BlockByBlockWithBufferedStream	4	10485760	64
WRITE	BlockByBlockWithBufferedStream	3	10485760	93
WRITE	BlockByBlockWithBufferedStream	2	10485760	137
WRITE	ByteByByteWithBufferedStream	0	10485760	188
WRITE	ByteByByteWithBufferedStream	0	10485760	228
WRITE	ByteByByteWithBufferedStream	0	10485760	233
WRITE	ByteByByteWithBufferedStream	0	10485760	222
WRITE	ByteByByteWithBufferedStream	0	10485760	217
WRITE	BlockByBlockWithoutBufferedStream	1000	10485760	40
WRITE	BlockByBlockWithoutBufferedStream	500	10485760	43
WRITE	BlockByBlockWithoutBufferedStream	50	10485760	355
WRITE	BlockByBlockWithoutBufferedStream	5	10485760	3440
WRITE	BlockByBlockWithoutBufferedStream	4	10485760	4298
WRITE	BlockByBlockWithoutBufferedStream	3	10485760	5684
WRITE	BlockByBlockWithoutBufferedStream	2	10485760	8454
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16450
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16450
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16599
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16512
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16480

operation	strategy	blockSize	fileSizeInBytes	durationInMs
READ	BlockByBlockWithBufferedStream	1000	10485760	9
READ	BlockByBlockWithBufferedStream	500	10485760	8
READ	BlockByBlockWithBufferedStream	50	10485760	10
READ	BlockByBlockWithBufferedStream	5	10485760	28
READ	BlockByBlockWithBufferedStream	4	10485760	29
READ	BlockByBlockWithBufferedStream	3	10485760	38
READ	BlockByBlockWithBufferedStream	2	10485760	57
READ	ByteByByteWithBufferedStream	0	10485760	65
READ	ByteByByteWithBufferedStream	0	10485760	65
READ	ByteByByteWithBufferedStream	0	10485760	63
READ	ByteByByteWithBufferedStream	0	10485760	62
READ	ByteByByteWithBufferedStream	0	10485760	62
READ	BlockByBlockWithoutBufferedStream	1000	10485760	14
READ	BlockByBlockWithoutBufferedStream	500	10485760	25
READ	BlockByBlockWithoutBufferedStream	50	10485760	214
READ	BlockByBlockWithoutBufferedStream	5	10485760	2106
READ	BlockByBlockWithoutBufferedStream	4	10485760	2635
READ	BlockByBlockWithoutBufferedStream	3	10485760	3642
READ	BlockByBlockWithoutBufferedStream	2	10485760	5297
READ	ByteByByteWithoutBufferedStream	0	10485760	10122
READ	ByteByByteWithoutBufferedStream	0	10485760	10122
READ	ByteByByteWithoutBufferedStream	0	10485760	10230
READ	ByteByByteWithoutBufferedStream	0	10485760	10306
READ	ByteByByteWithoutBufferedStream	0	10485760	10159





Analyse des mesures

Avec ces différentes mesures, nous remarquons bien sûr que l'utilisation de tampons permet d'améliorer significativement les temps d'écriture et de lecture des données. Que ce soit avec ou sans tampons, nous remarquons également que l'utilisation de blocs de données de plus en plus grands (en écriture et en lecture), nous donne des performances de plus en plus grandes. Si nous comparons l'écriture et la lecture des données, il y a une plus forte tendance à ce que les temps se rapprochent en lecture avec ou sans utilisation de tampons. Enfin, nous remarquons que l'utilisation de tampons même avec une taille de bloc égale à 0 donne des temps bien plus petits que sans l'utilisation de tampons que ce soit en écriture ou lecture.

Modification du code du programme de base

Le programme qui nous était fourni au début du laboratoire ne permettait pas encore d'enregistrer les différents temps obtenus lors de l'expérience dans un fichier au format « csv ». Nous avons dû le modifier pour pouvoir le faire. Nous avons donc rajouté trois interfaces supplémentaires ainsi que trois classes implémentant ces trois interfaces. La première classe « ExperimentData » permet d'enregistrer les données que nous voulons écrire dans le fichier « csv » (comme le temps, la taille du fichier, la stratégie, …) dans une map (avec une association entre une clé et une valeur). Ces données correspondent donc aux données générées durant l'expérience. La deuxième classe « FileRecorder » permet de créer le fichier de sortie au format « csv » ainsi que le flux de sortie (avec tampon) connecté au fichier de sortie. Enfin, la troisième classe « CsvSerializer » permet de sérialiser les données de l'expérience (récupérées depuis l'objet de la première classe) au format « csv » et de les écrire dans notre fichier de sortie. Le format « csv » correspond aux différentes données séparées par des « , ». Après avoir créé ces trois classes, nous les avons utilisées dans la classe principale du programme. Dans cette dernière, nous avons rajouté un constructeur dans lequel nous initialisons un objet pour la sérialisation (troisième classe) et un autre pour créer le fichier « csv » voulu (deuxième classe). Finalement, nous créons un nouvel objet de données (première classe) à chaque fois que nous appelons la méthode pour écrire des données de test dans des fichiers et celle pour lire ces mêmes données dans ces mêmes fichiers. Ci-dessous, le code permettant d'enregistrer les données de l'expérience dans un fichier « csv » est présenté (pour l'écriture de données).

```
// add data to the data object
data.addData("WRITE", ioStrategy.toString(), blockSize, NUMBER_OF_BYTES_TO_WRITE, time);
// record data added in the csv file
recorder.record(data);
```